D4

(19) Europäisches Patentamt
European Patent Office
Office européen des brevets

(11) EP 0 723 234 A1

(12) EUROPEAN PATENT APPLICATION

(43) Date of publication:
24.07.1996 Bulletin 1996/30

(51) Int. Cl.^{6}: G06F 15/16, G06F 13/10

(21) Application number: 95114570.5

(22) Date of filing: 15.09.1995

(84) Designated Contracting States:
DE FR GB

(30) Priority: 19.01.1995 US 376179

(71) Applicant: Hewlett-Packard Company
Palo Alto, California 94304 (US)

(72) Inventors:
- Larson, Susan L.
Kuna, Idaho 83634 (US)
- Voigt, Douglas L.
Boise, Idaho 83702 (US)
- Messinger, Steven D.
Boise, Idaho 83713 (US)
- Jacobson, Michael B.
Boise, Idaho 83714 (US)

(74) Representative: Schoppe, Fritz, Dipl.-Ing.
Patentanwalt,
Georg-Kalb-Strasse 9
D-82049 Pullach (DE)

(54) Data storage system and method for managing asynchronous attachment and detachment of storage disks

(57) A disk array data storage system (14) has plural sets of storage disks (70-80) connected to multiple corresponding I/O buses (64-67). Individual storage disks can be independently and asynchronously attached to or detached from corresponding interfacing slots (50-61) of the I/O buses. The system has physical device drivers (92) which represent the storage disks with respect to their connections to the interfacing slots of the I/O buses and physical device managers (94) which represent the data kept on the storage disks. Interface drivers (90) are provided to manage I/O transfers through corresponding I/O buses. When a particular storage disk (80) is attached to or detached from an interfacing slot (55) of an I/O bus (65), the interface driver (90) corresponding to the I/O bus freezes all I/O requests that come from the physical device drivers (92) that represent the storage disks in the set connected to the I/O bus. A configuration manager (96) determines which interfacing slot (55) the particular storage disk has been attached to or detached from. If the particular storage disk has been detached from the I/O bus, the configuration manager (96) eliminates the physical device driver (92) that represents the detached storage disk. The data stored on the detached storage disk is rebuilt using redundancy on the remaining disks. If the particular storage disk has been attached to the I/O bus, the configuration manager (96) creates a new physical device driver (92) to represent the attached storage disk. The I/O requests to the I/O bus are then unfrozen.

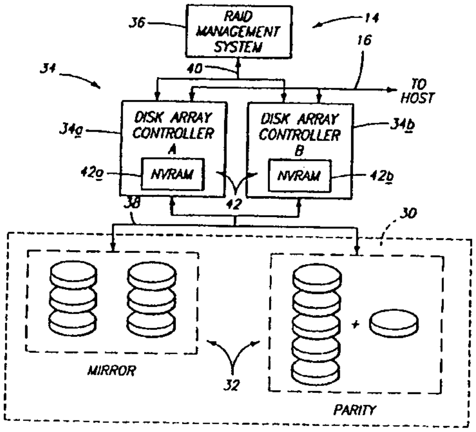


Fig. 2

Printed by Rank Xerox (UK) Business Services
2 13.0/3.4

Description

FIELD OF THE INVENTION

This invention relates to disk array data storage systems, and more particularly, to systems and methods for managing asynchronous attachment and detachment of independent storage disks.

BACKGROUND OF THE INVENTION

Disk array data storage systems have multiple storage disk drive devices which are arranged and coordinated to form a single mass storage system. The disk array has multiple mechanical bays or interfacing slots which receive individual storage disks. The memory capacity of such a storage system can be expanded by adding more disks to the system, or by exchanging existing disks for larger capacity disks.

Some conventional disk array data storage systems permit a user to connect or "hot plug" additional storage disks to available interfacing slots while the system is in operation. Although the "hot plug" feature is convenient from a user standpoint, it presents some difficult control problems for the data storage system. One problem that arises during "hot plug" occurs when a user attempts to switch storage disks among the various interfacing slots. For instance, a user might remove the disk that was in one interfacing slot and plug it into another interfacing slot. When this occurs, the conventional disk array system writes data to the incorrect storage disk. The prior art solution to this problem has been to simply place a limitation on the user not to shuffle the storage disks among interfacing slots. Thus, once a storage disk is assigned to a slot, it remains there.

Another problem concerns the processing of I/O requests during removal of a storage disk from the disk array. When the storage disk is decoupled from the interfacing slot, conventional storage systems typically return a "timeout" warning, indicating that the I/O request has failed. This warning is usually generated when the storage system can not access an existing storage disk due to mechanical or other storage problems. Here, in contrast, the reason for the failed access is that the storage disk is missing. A more appropriate interpretation, then, is to report that the storage disk has been removed, not that the I/O request has failed. It would be helpful if a disk array could distinguish between a missing storage disk and a failed I/O request to an existing storage disk.

Another problem facing large disk array data storage systems having many independent storage disks concerns the effect that attachment or detachment of a storage disk has on the rest of the storage disks coupled to the system. In present disk arrays, attachment or removal of a storage disk causes a temporary halt of all I/O activity to all storage disks on the disk array. It would be advantageous to construct a disk array that identifies the storage disk that has been attached or removed and to isolate I/O activity involving that storage disk from other I/O's, thus enabling the remaining storage disks to continue activity.

SUMMARY OF THE INVENTION

A disk array data storage system according to an aspect of this invention has multiple sets of plural storage disks connected to multiple corresponding I/O buses. Individual storage disks can be independently and asynchronously attached to or detached from corresponding interfacing slots of the I/O buses. The system has physical device drivers which represent the storage disks with respect to their connections to the interfacing slots of the I/O buses. The system further includes physical device managers which represent the data kept on the storage disks.

Interface drivers are provided to manage I/O transfers through corresponding I/O buses. When a particular storage disk is attached to or detached from an interfacing slot of an I/O bus, the I/O bus is reset. This stops activity on the I/O bus and prevents on-going activity from completing. The interface driver corresponding to the I/O bus freezes all I/O requests to the physical device drivers that represent the storage disks in the set connected to the I/O bus.

A configuration manager is provided to determine which interfacing slot the particular storage disk has been attached to or detached from. If a storage disk has been detached from the I/O bus, the configuration manager eliminates the physical device driver that represents the detached storage disk. The missing data stored on the detached storage disk is then rebuilt on the remaining disks using redundant data. Thereafter, the physical device manager for the detached disk can be eliminated.

If a new storage disk has been attached to the I/O bus, the configuration manager creates a new physical device driver to represent the attached storage disk with respect to its connection to the interfacing slot. A new physical device manager may also be created to represent the data on the new storage disk.

The configuration manager reverifies all storage disks connected to the I/O bus and then signals the interface driver causing it to flush the I/O requests to the physical device drivers for appropriate action. By initially freezing the queues of I/O requests to the I/O bus and then reverifying all storage disks attached to the I/O bus, including the newly attached disk, the system effectively prohibits user commands from accessing the wrong storage disks. Additionally, the system quickly identifies and isolates the newly attached/detached storage disk so that the I/O activity to the remaining storage disks on the disk array can continue.

According to other aspects of this invention, methods for asynchronously attaching and detaching a storage disk to and from a disk array data storage system are provided.

DESCRIPTION OF THE DRAWINGS

Preferred embodiments of the invention are described below with reference to the following accompanying drawings depicting examples embodying the best mode for practicing the invention.

Fig. 1 is a diagrammatical illustration of a host computer station connected to a disk array data storage system of this invention.

Fig. 2 is a block diagram of a data storage system embodied as a hierarchical disk array.

Fig. 3 is a block diagram illustrating the interfacing and management of independent storage disks.

Fig. 4 is a flow diagram of general steps for asynchronously attaching and detaching a storage disk according to an aspect of this invention.

Fig. 5 is a process flow diagram illustrating preferred control steps for asynchronously attaching a storage disk to a disk array.

Fig. 6 is a process flow diagram illustrating preferred control steps for asynchronously detaching a storage disk from a disk array.

DETAILED DESCRIPTION OF THE INVENTION

This disclosure of the invention is submitted in furtherance of the constitutional purposes of the U.S. Patent Laws "to promote the progress of science and useful arts". U.S. Constitution, Article 1, Section 8.

Fig. 1 shows a computer system 10 having a host computer terminal or station 12 connected to a data storage system 14 via host interface bus 16. Host computer station 12 includes a visual display monitor 18, a central processing unit (CPU) 20, and a keyboard 22.

Fig. 2 shows an example construction of data storage system 14 embodied as a redundant hierarchic disk array data storage system. Disk array storage system 14 includes a disk array 30 having a plurality of storage disks 32, a disk array controller 34 coupled to the disk array 30 to coordinate data transfer to and from the storage disks 32, and a RAID management system 36.

In this example construction, disk array controller 34 is implemented as a dual controller consisting of disk array controller A 34a and disk array controller B 34b. Dual controllers 34a and 34b enhance reliability by providing continuous backup and redundancy in the event that one controller becomes inoperable. The disk array controller 34 is coupled to the host computer via host interface bus 16.

RAID management system 36 is operatively coupled to disk array controller 34 via an interface protocol 40. The term "RAID" (Redundant Array of Independent Disks) means a disk array in which part of the physical storage capacity is used to store redundant information about user data stored on the remainder of the storage capacity. The redundant information enables regeneration of user data in the event that one of the array's member disks or the access path to it fails. A more detailed discussion of RAID systems is found in a book entitled, *The RAIDBook: A Source Book for RAID Technology*, published June 9, 1993, by the RAID Advisory Board, Lino Lakes, Minnesota.

RAID management system 36 can be embodied as a separate component, or configured within disk array controller 34 or within the host computer to provide a data manager means for controlling disk storage and reliability levels, and for transferring data among various reliability storage levels. These reliability storage levels are preferably mirror or parity redundancy levels, but can also include a reliability storage level with no redundancy at all.

Redundant hierarchic disk array 30 can be characterizable as different storage spaces, including its physical storage space and one or more virtual storage spaces. These various views of storage are related through mapping techniques. For example, the physical storage space of the disk array can be mapped into a RAID-level virtual storage space which delineates storage areas according to the various data reliability levels. For instance, some areas within the RAID-level virtual storage space can be allocated for a first reliability storage level, such as mirror or RAID level 1, and other areas can be allocated for a second reliability storage level, such as parity or RAID level 5. The RAID-level virtual view can be mapped to a second application-level virtual storage space which presents a contiguously addressable storage space. The physical configuration and RAID view of the storage space are hidden from the application view, which is presented to the user.

A memory map store 42 provides for persistent storage of the virtual mapping information used to map different storage spaces into one another. The memory mapping information can be continually or periodically updated by the controller or RAID management system as the various mapping configurations among the different views change. In this configuration, memory map store 42 is embodied as two non-volatile RAMs (Random Access Memory) 42a and 42b, such as battery-backed RAMs, which are located in respective controllers 34a and 34b. The dual NVRAMs 42a and 42b provide for redundant storage of the memory mapping information.

Fig. 3 shows a preferred architecture of disk array data storage system 14. Disk array controller 34 is coupled to the disk array via multiple internal I/O buses, referenced generally by numeral 38. Preferably, each I/O bus is a small computer system interface (SCSI) type bus. Each I/O bus is operably connected to a set of storage disks. In this example construction, the disk array data storage system can accommodate twelve storage disks and thus has twelve active mechanical bays or interfacing slots 50-61. Four SCSI I/O buses 64-67 are coupled to respective sets of three interfacing slots. That is, I/O bus 64 is coupled to interfacing slots 50-52; I/O bus 65 is coupled to interfacing slots 53-55; I/O bus 66 is coupled to interfacing slots 56-58; and I/O bus 67 is coupled to interfacing slots 59-61.

The storage disks are independently detachably connected to I/O buses 64-67 at interfacing slots 50-61. Each storage disk can be asynchronously attached to or detached from the interfacing slots. The disk array data storage system is shown as having ten existing storage disks 70-79 that are detachably connected to interfacing slots 50-54 and 56-60, respectively. Once interfacing slot 61 is open and available to receive a storage disk. Storage disk 80 is shown as being attached to or detached from interfacing slot 55. The storage disks have example sizes of one to three Gigabytes. If all slots are filled, the data storage system has an example combined capacity of 12-36 Gigabytes.

Interface drivers 90 are provided to manage I/O transfers through I/O buses 64-67. There are preferably four interface drivers, one for each bus. The interface drivers 90 are implemented in firmware resident in disk array controller 34.

The disk array data storage system of this invention includes multiple physical device drivers 92 and multiple physical device managers 94. There are preferably twelve physical device drivers, one for each storage disk that can be connected to the disk array. The physical device drivers 92 are implemented as objects in firmware to represent the storage disks with respect to their connections to the interfacing slots of the I/O buses. For example, one physical device driver represents whatever disk is connected to interfacing slot 52 of I/O bus 64, which in this case, is storage disk 72.

The physical device managers 94 are implemented as objects in firmware to represent data kept on the individual storage disks. There are preferably sixteen physical device managers, one for each storage disk, plus an extra four to temporarily represent data on any disks which have recently been removed from the system. The physical device managers are associated with corresponding physical device drivers that represent the same storage disks. During a normal read/write request from the host, the request is passed through the physical device manager 94, to the physical device driver 92, to the interface driver 90, and to the appropriate I/O bus and storage disk.

A configuration manager 96 is also provided to manage operation of the disk array data storage system in the event a storage disk is attached to or detached from an I/O bus. The configuration manager is implemented as an object in firmware to schedule events that control the conduct of the physical device managers and physical device drivers that represent the newly attached/detached storage disk.

For purposes of continuing discussion, suppose that storage disk 80 in Fig. 3 is being attached to or detached from interfacing slot 55. The action of attaching or detaching a storage disk causes generation of a reset condition according to conventional electromechanical techniques. The interfacing electromechanics of the storage disk and slot involve connector pins of different lengths (typically 2-3 different lengths). During attachment, the longer pins make initial contact, followed by the shorter pins, until the drive is connected. A reset condition is generated and placed on I/O bus 65 after the longer pins are inserted. The reset condition on the I/O bus goes away after the shorter pins are inserted to complete connection. The reset condition induced by the mechanical multi-tier insertion scheme is sensed in the disk array controller and causes an interrupt within the controller. Reset halts the I/Os in progress on the affected bus.

The reverse process is used for detachment. The shorter pins are first disconnected, followed by the longer pins. This mechanical event causes a reset condition indicative of disk removal.

Fig. 4 shows a general method for asynchronously attaching and detaching a storage disk to and from a disk array data storage system according to this invention. At step 100, storage disk 80 is detected as being attached to or detached from the interfacing slot 55 on I/O bus 65. Upon detection, the queues of I/O requests to I/O bus 65 are frozen (step 102). It is noted that the I/O requests to the other three I/O buses 64, 66, and 67 continue to be processed.

At step 104, the disk array controller distinguishes the newly attached/detached storage disk 80 from the other storage disks 73 and 74 that are connected to the same I/O bus 65. Once the specific disk and interfacing slot are identified, if storage disk 80 has been detached, the configuration manager starts the deletion of the physical device driver for that disk (step 105). Then, the queues of I/O requests to the other storage disks 73 and 74 are unfrozen to permit their continued use (step 106). In this manner, the method of this invention effectively identifies and isolates the activity of only the single storage disk that has been attached or detached. Access to the other disks continues with minimal interruption. After the queues are unfrozen, if storage disk 80 has been attached, the configuration manager creates a physical device driver for that disk and requests that the physical device driver make the disk ready for use. When storage disk 80 is ready for use, the configuration manager associates the physical device driver with a physical device manager.

The methods for asynchronous attachment and detachment of storage disks will now be described separately below, but in more detail, with reference to Figs. 3, 5 and 6.

Method for Asynchronous Attachment

Fig. 5 illustrates preferred steps for a method for asynchronously attaching a storage disk to a disk array data storage system. For this discussion, assume that storage disk 80 is being attached to slot 55 of I/O bus 65. Fig. 5 shows an interface driver (ID) 90 for I/O bus 65, a physical device driver (PDD) 92 that represents storage disk 80 with respect to its connection to slot 55, a physical device manager (PDM) 94 that represents the data kept on storage disk 80, and a configuration manager (CFM) 96. The process is labelled alphanumerically as steps A1-A16, with the letter "A" designating the attachment process. A shorthand description for each step is provided in the depicted table.

At step A1, a reset condition is generated by the electro-mechanical interface structure upon insertion of storage disk 80 into interfacing slot 55. The reset condition is sent over I/O bus 65 to the interface driver (ID). The I/O bus 65 is thus immediately identified as being the bus to which a new storage disk is attached, and not the other three buses 64, 66, and 67. At step A2, the interface driver (ID) freezes all queues of I/O requests to I/O bus 65. The interface driver continues to receive I/O requests, but does not process them. I/Os in progress and I/Os that arrive are frozen in the interface driver (ID) queues. The physical device driver (PDD) maintains its own queues which are not frozen. It can still receive I/Os from the configuration manager (CFM) and the physical device manager (PDM) and forward I/O requests to the interface driver (ID). The interface driver (ID), however, will not forward the I/O requests to the storage disks.

At step A3, the interface driver (ID) notifies the configuration manager (CFM) that a reset condition has been detected in I/O bus 65. The configuration manager (CFM) examines all three interface slots 53-55 supported by I/O bus 65 to distinguish the new storage disk 80 from the other existing storage disks 73 and 74. The identification process includes scanning for each storage disk attached to I/O bus 65 (step A4) by repeatedly requesting and receiving identification information via the interface driver (ID) from each storage disk (steps A5 and A6). After all three interface connections have been scanned, the configuration manager can make specific identification of the newly attached storage disk 80 (step A7).

At step A8, the configuration manager (CFM) notifies the interface driver (ID) to unfreeze the queues of I/O requests to the other storage disks 73 and 74 on I/O bus 65. The interface driver (ID) informs the physical device driver (PDD) of this queue flush condition with hot reset status (step A9). The hot reset status is simultaneously sent by the interface driver to other physical device drivers (not shown) that represent attached storage disks 73 and 74, causing the disks to be reconfigured. After this point, control communication flow is carried on primarily between the configuration manager (CFM) and the physical device driver (PDD).

At step A10, the configuration manager (CFM) creates a new physical device driver (PDD) to represent the newly attached storage disk 80 at interfacing slot 55 on I/O bus 65. The new physical device driver (PDD) is initialized with the channel or I/O bus number, and the interfacing slot number (step A11). The configuration manager (CFM) then instructs the physical device driver (PDD) to "spin up" or begin operation of the new storage disk 80 (step A12). The physical device driver (PDD) returns the unique vendor serial number of storage disk 80 to the configuration manager (CFM) (step A13) and commences spin up of the new storage disk (step A14). Once the storage disk 80 is determined to be ready, it is configured for correct operation in the disk array.

It is noted that simultaneous to steps A10-A14 for the newly added storage disk 80, the configuration manager (CFM) is preferably reverifying the identity and location of the other two storage disks 73 and 74 that are also connected to I/O bus 65. When the physical device drivers (not shown) for the storage disks 73 and 74 receive the hot reset status or SCSI unit attention, they automatically reconfigure these storage disks. As part of the configuration, the physical device driver (PDD) retrieves the serial number from the storage disk and reports it to the configuration manager (CFM). In this manner, the configuration manager (CFM) essentially revalidates the serial numbers of all three storage disks on the I/O bus to detect any new changes resulting from the attachment of the new storage disk, such as removal and insertion of a storage disk into the same slot.

At step A15, the physical device driver (PDD) informs the configuration manager (CFM) that the spin up and configuration of newly attached storage disk 80 is complete. At that point, the configuration manager can create a new physical device manager (PDM) which represents the data on newly attached storage disk 80 (step A16). This step entails initializing the variables of the physical device manager (PDM) to uniquely represent the particular storage disk 80.

Alternatively, the configuration manager (CFM) might associate the physical device driver (PDD) with an existing physical device manager (PDM). This is the situation, for example, when the new storage disk 80 had been previously attached to the system in another interfacing slot and a physical device manager (PDM) which represents the data content on that disk already exists. Recall that the data storage system can have sixteen physical device managers (PDM) for a potential twelve storage disks. The extra four physical device managers (PDM) can continue to represent data on the four most recently detached storage disks.

Once a physical device manager (PDM) is assigned to storage disk 80, the space on storage disk 80 is now made available to the user. If the newly inserted disk is deemed inoperable or ineligible for normal access, a placeholder physical device manager (PDM) is created to present the disk to the host for diagnosis.

It is noted that the system and method according to this invention effectively solves the prior art problem of writing data to a wrong storage disk that is discussed above in the Background of the Invention section. The system and method of this invention call for immediately freezing the queues of I/O requests to the I/O bus that generated the reset condition. Thereafter, the storage disks attached to the I/O bus, including the newly attached bus, are reverified to double check which storage disks are attached to the I/O bus before the I/O requests are processed. As a result, it is impossible for the user commands to reach the wrong storage disks.

Method for Asynchronous Detachment

Fig. 6 illustrates preferred steps for a method for asynchronously detaching a storage disk from a disk array data storage system according to an aspect of this invention. For this discussion, assume that storage disk 80 is being removed from slot 55 of I/O bus 65. Fig. 6 shows the interface driver (ID) 90 for I/O bus 65, the physical device driver (PDD) 92 that represents storage disk 80 with respect to its connection to slot 55, the physical device manager (PDM) 94 that represents the data kept on storage disk 80, and the configuration manager (CFM) 96. The process is labelled alphanumerically as steps D1-D15, with the letter "D" designating the detachment process. A shorthand description for each step is provided in the depicted table.

Steps D1-D7 are essentially the same as steps A1-A7 described above. Briefly, at step D1, a reset condition is generated by the electro-mechanical interface structure, upon removal of storage disk 80 from interfacing slot 55, and sent over I/O bus 65 to the interface driver (ID). At step D2, the interface driver (ID) freezes all queues of I/O requests to I/O bus 65. At step D3, the interface driver (ID) notifies the configuration manager (CFM) that a reset condition has been detected in I/O bus 65. The configuration manager (CFM) examines all three interface slots 53-55 supported by I/O bus 65 to identify the detached storage disk 80 from among the other existing storage disks 73 and 74 (steps D4-D7).

At step D8, the configuration manager (CFM) notifies the physical device manager (PDM) that its represented storage disk 80 has been detached from the disk array. The physical device manager (PDM) is instructed to stop submitting I/O requests to its associated physical device driver (PDD) which represents the same detached storage disk. At step D9, the configuration manager (CFM) initiates a process to eliminate the physical device driver (PDD) that represents the detached storage disk 80. The physical device driver (PDD) places itself in a delete-in-progress state and awaits the flushing of any final I/O requests from the interface driver (steps D10 and D11).

At step D12, the configuration manager (CFM) notifies the interface driver (ID) to unfreeze the queues of I/O requests to the storage disks on I/O bus 65. The interface driver (ID) informs the physical device driver (PDD) of this queue flush condition (step D13). After the active queue is emptied, the physical device driver (PDD) notifies the configuration manager (CFM) that the deletion is complete (steps D14 and D15).

All unanswered I/O requests are returned to the associated physical device manager (PDM) which already knows that the storage disk is missing. The data is reconstructed using the RAID management system in order to complete the I/O request. For redundant disk arrays, such as the example RAID system described above in Fig. 2, the missing data on the removed storage disk 80 can be rebuilt on other storage disk using the redundant data. The physical device manager (PDM) for the detached storage disk remains viable until the rebuild process is finished in order to track deallocation of data from the missing drive.

Although the processes of this invention have been described separately, multiple attachments and/or detachments can occur at the same time involving one or more buses. The same procedures described above apply in these situations as well.

It is noted that the system and method according to this invention provides the user with an accurate description of why an I/O request may not be immediately processed. When a storage disk is removed, the physical device driver (PDD) for the storage disk is deleted (step D9 and D10) before the I/O requests are flushed (step D13). The system therefore returns a warning to the user indicating that a storage disk is missing, but continues to process I/O requests using redundant data as described above. Therefore, unlike prior art systems, the system of this invention does not return a misleading "timeout" notice indicating that the request is denied due to a faulty drive or other I/O problem.

Another advantage of this invention is that the attached/detached storage disk is quickly identified and isolated so that the I/O activity to the remaining storage disks on the disk array can continue. This is an improvement over prior art systems that temporarily halt all I/O activity to all storage disks on the disk array.

The combined use of associated physical device drivers and physical device managers provide further benefits of this invention. A user is now free to move disks from one interfacing slot to another without limitation. Upon withdrawal of a storage disk from the first slot, the physical device driver representing the storage disk in the first slot is deleted. The physical device manager for the storage disk remains. When the disk is reinserted into the second slot, a new physical device driver representing the storage disk in the second slot is created. The new physical device driver is then associated with the physical device manager for the same disk, and all is ready for normal operation.

The system and methods of this invention therefore provide tremendous flexibility to the user. The user can add or replace disk drives or rearrange their connection to the disk array without affecting data content and reliability. Additionally, all of the data remains accessible during the "hot plug" process.

In compliance with the statute, the invention has been described in language more or less specific as to structural and methodical features. It is to be understood, however, that the invention is not limited to the specific features shown and described, since the means herein disclosed comprise preferred forms of putting the invention into effect. The invention is, therefore, claimed in any of its forms or modifications within the proper scope of the appended claims appropriately interpreted in accordance with the doctrine of equivalents.

Claims

1. A method for asynchronously attaching and detaching a storage disk (80) to and from a disk array data storage system (14), the disk array data storage system comprising multiple storage disks (73, 74) connected to at least one internal I/O bus (65), the method comprising the following steps:

   detecting when a particular storage disk (80) is attached to or detached from an I/O bus (65) in a disk array data storage system (14), the disk array data storage system having other storage disks connected to the I/O bus (73, 74);

   freezing queues of I/O requests to the I/O bus (65) that said particular storage disk (80) is attached to or detached from;

   distinguishing said particular storage disk (80) from the other storage disks (73, 74) connected to the I/O bus (65); and

   unfreezing the queues of I/O requests to the other storage disks (73, 74) on the I/O bus.

2. A method according to claim 1 further comprising the following additional step:

   reverifying identity and location of all storage disks connected to the I/O bus.

3. A method according to claim 2 wherein the storage disks have associated serial numbers, the reverifying step comprising the following additional steps:

   reconfiguring all storage disks connected to the I/O bus; and

   revalidating the serial numbers of the storage disks connected to the I/O bus.

4. A method according to claim 1 wherein the disk array data storage system has multiple I/O buses (64-67), individual I/O buses being capable of connecting with multiple storage disks (70-80), the method further comprising the following additional steps:

   identifying the I/O bus (65) which said particular storage disk (80) has been attached to or detached from; and

   distinguishing said particular storage disk (80) from the other storage disks (73, 74) connected to the same identified I/O bus (65).

5. A method for asynchronously attaching a storage disk to a disk array data storage system, the disk array data storage system (14) comprising a disk array controller (34), plural storage disks (70-80), and multiple internal I/O buses (64-67), individual I/O buses being capable of interconnecting and transferring I/O requests between the disk array controller and multiple storage disks, the method comprising the following steps:

   providing physical device drivers within the disk array controller (34) which represent the storage disks with respect to their connections to the I/O buses;

   detecting when a new storage disk (80) is attached to one of the I/O buses (65) in the disk array data storage system;

   freezing queues of I/O requests to said one I/O bus (65) to which the new storage disk (80) is attached;

   distinguishing the new storage disk (80) from any other storage disks (73, 74) connected to said one I/O bus (65);

   unfreezing the queues of I/O requests to the other storage disks (73, 74) connected to said one I/O bus (80);

   reverifying identity and location of all storage disks connected to said one I/O bus, including the new storage disk; and

   creating a new physical device driver (92) for the new storage disk.

6. A method according to claim 5 further comprising the following additional steps:

   providing physical device managers within the disk array controller (34) which represent data kept on the storage disks; and

   associating the new physical device driver (92) with a physical device manager (94).

7. A method for asynchronously detaching a storage disk from a disk array data storage system, the disk array data storage system (14) comprising a disk array controller (34), plural storage disks (70-80), and multiple internal I/O buses (64-67), individual I/O buses being capable of interconnecting and transferring I/O requests between the disk array controller and multiple storage disks, the method comprising the following steps:

   providing physical device drivers within the disk array controller (34) which represent the storage disks with respect to their connections to the I/O buses;

   providing physical device managers within the disk array controller which represent data kept on the storage disks, the physical device managers being associated with corresponding physical device drivers that represent the same storage disks;

   detecting when an existing storage disk (80) is detached from one of the I/O buses (65) in the disk array data storage system;

   freezing queues of I/O requests to said one I/O bus (65) from which the existing storage disk (80) is detached;

   distinguishing the detached storage disk (80) from any other storage disks (73, 74) connected to said one I/O bus;

   informing the physical device manager (94) that represents the detached storage disk (80) that the detached storage disk is disconnected from said one I/O bus (65);

eliminating within the disk array controller (34) the physical device driver (92) that represents the detached storage disk (80); and

unfreezing the queues of I/O requests to said one I/O bus (65).

8. A method according to claim 7 further comprising the following additional steps:

storing data redundantly on the storage disks; and

rebuilding missing data on the detached storage disk (80) onto the storage disks (70-79) in the disk array data storage system using the redundant data stored on the storage disks.

9. A disk array data storage system comprising:

a plurality of storage disks (70-80);

multiple I/O buses (64-67), individual I/O buses being connected to a set of storage disks;

the storage disks being detachably connected to the I/O buses at interfacing slots (50-61) whereby individual storage disks can be asynchronously attached to or detached from corresponding interfacing slots of the I/O buses;

a plurality of physical device drivers (92) for corresponding storage disks, the physical device drivers representing the storage disks with respect to their connections to the interfacing slots (50-61) of the I/O buses (64-67);

a plurality of physical device managers (94) for corresponding storage disks, the physical device managers representing data kept on the storage disks, the physical device managers being associated with corresponding physical device drivers (94) that represent the same storage disks;

interface drivers (90) to manage I/O transfers through corresponding I/O buses;

when a particular storage disk (80) is attached to or detached from an interfacing slot (55) of one of the I/O buses (65), the interface driver (90) corresponding to said one I/O bus freezing I/O requests that came from the physical device drivers (92) that represent the storage disks (73, 74, 80) in the set connected to said one I/O bus (65);

a configuration manager (96) to manage operation of the disk array data storage system when said particular storage disk (80) is attached to or detached from said one I/O bus (65), the configuration manager (96) first determining which interfacing slot (55) the particular storage disk has been attached to or detached from and then subsequently instructing the interface driver (90) to unfreeze the I/O requests and flush the I/O requests back to the physical device drivers associated with the other storage disks (73, 74) in the set of storage disks connected to said one I/O bus (65).

10. A disk array data storage system according to claim 9 wherein:

upon removal of said particular storage disk (80) from said one I/O bus (65), the configuration manager (96) eliminates the physical device driver (92) that represents said particular storage disk; and

upon attachment of said particular storage disk (80) to said one I/O bus (65), the configuration manager (96) creates a new physical device driver (92) to represent said particular storage disk (80) with respect to its connection to the interfacing slot (55) of said one I/O bus (65).

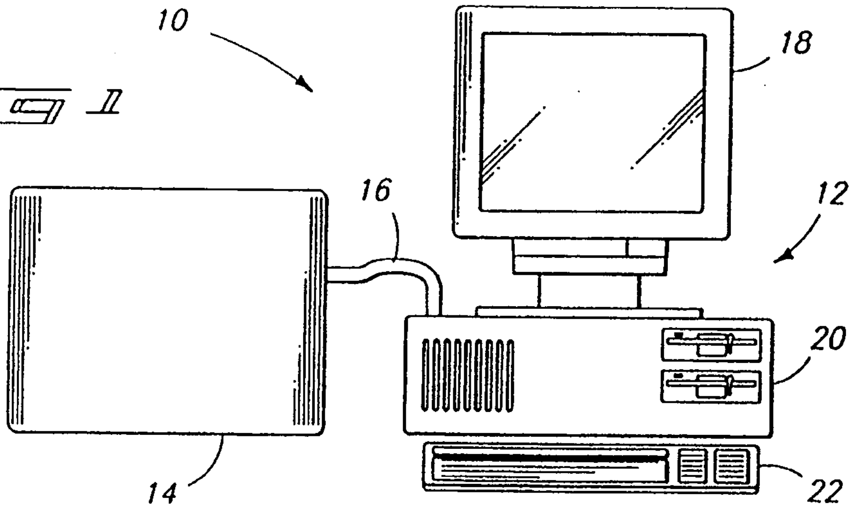

10
18
16
12
20
14
22

Fig 1

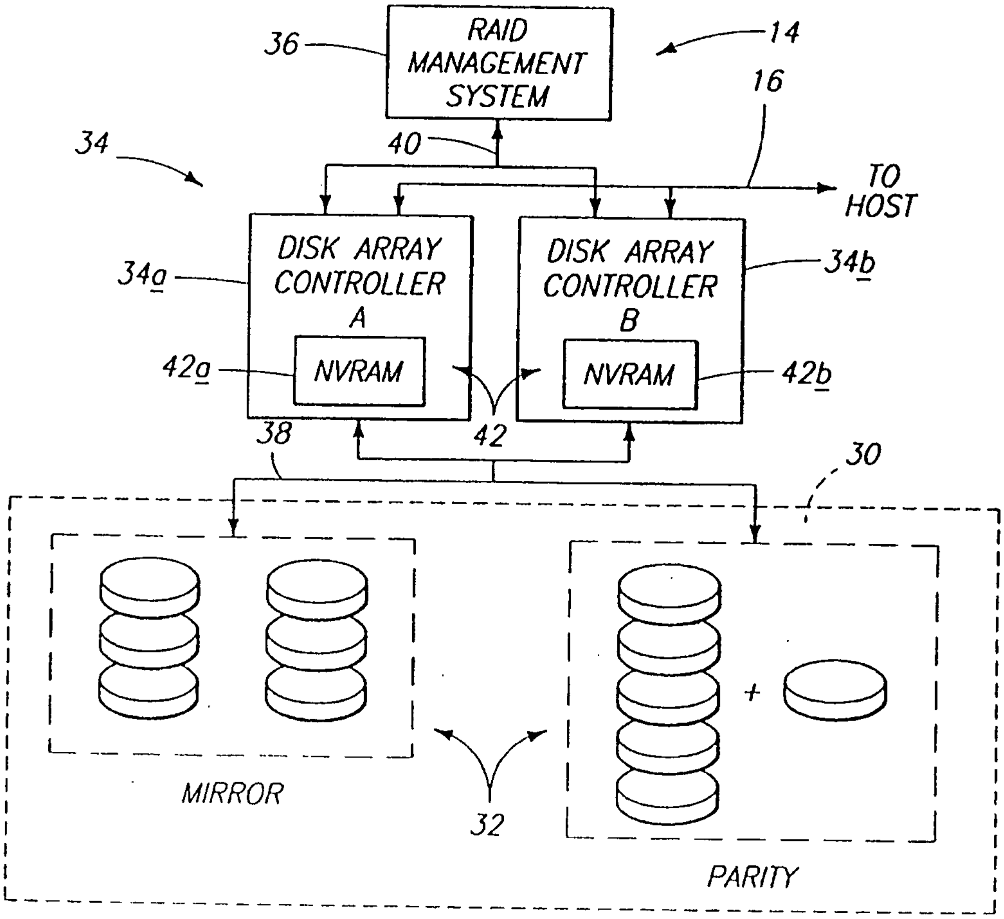

RAID MANAGEMENT SYSTEM
36
14
16
34
40
TO HOST
DISK ARRAY CONTROLLER A
DISK ARRAY CONTROLLER B
34a
34b
NVRAM
NVRAM
42a
42b
42
38
30
+
MIRROR
32
PARITY

Fig 2

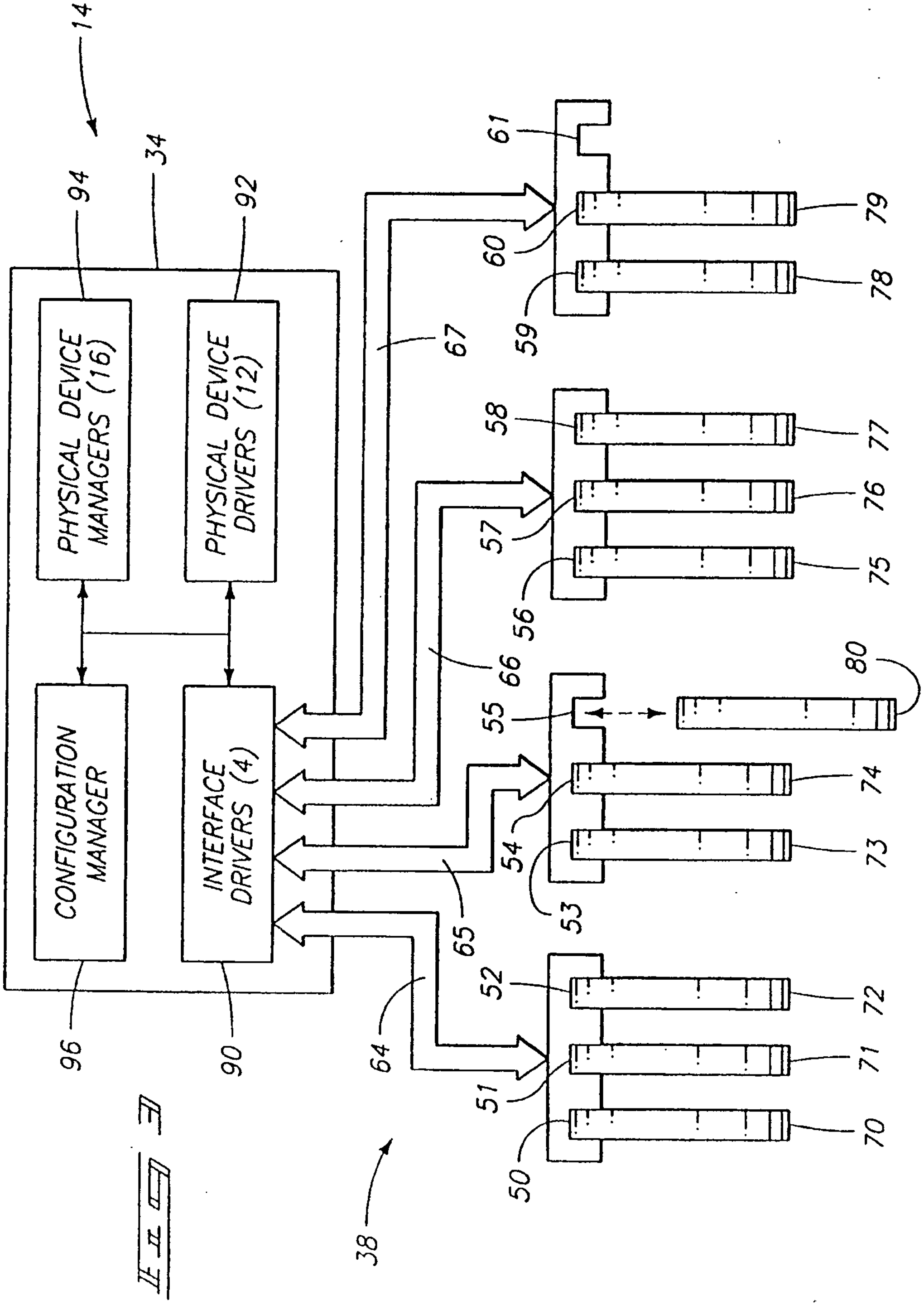

14
34
94
92
PHYSICAL DEVICE MANAGERS (16)
PHYSICAL DEVICE DRIVERS (12)
CONFIGURATION MANAGER
INTERFACE DRIVERS (4)
96
90
64
65
66
67
38
50
51
52
53
54
55
56
57
58
59
60
61
70
71
72
73
74
75
76
77
78
79
80

FIG. 2

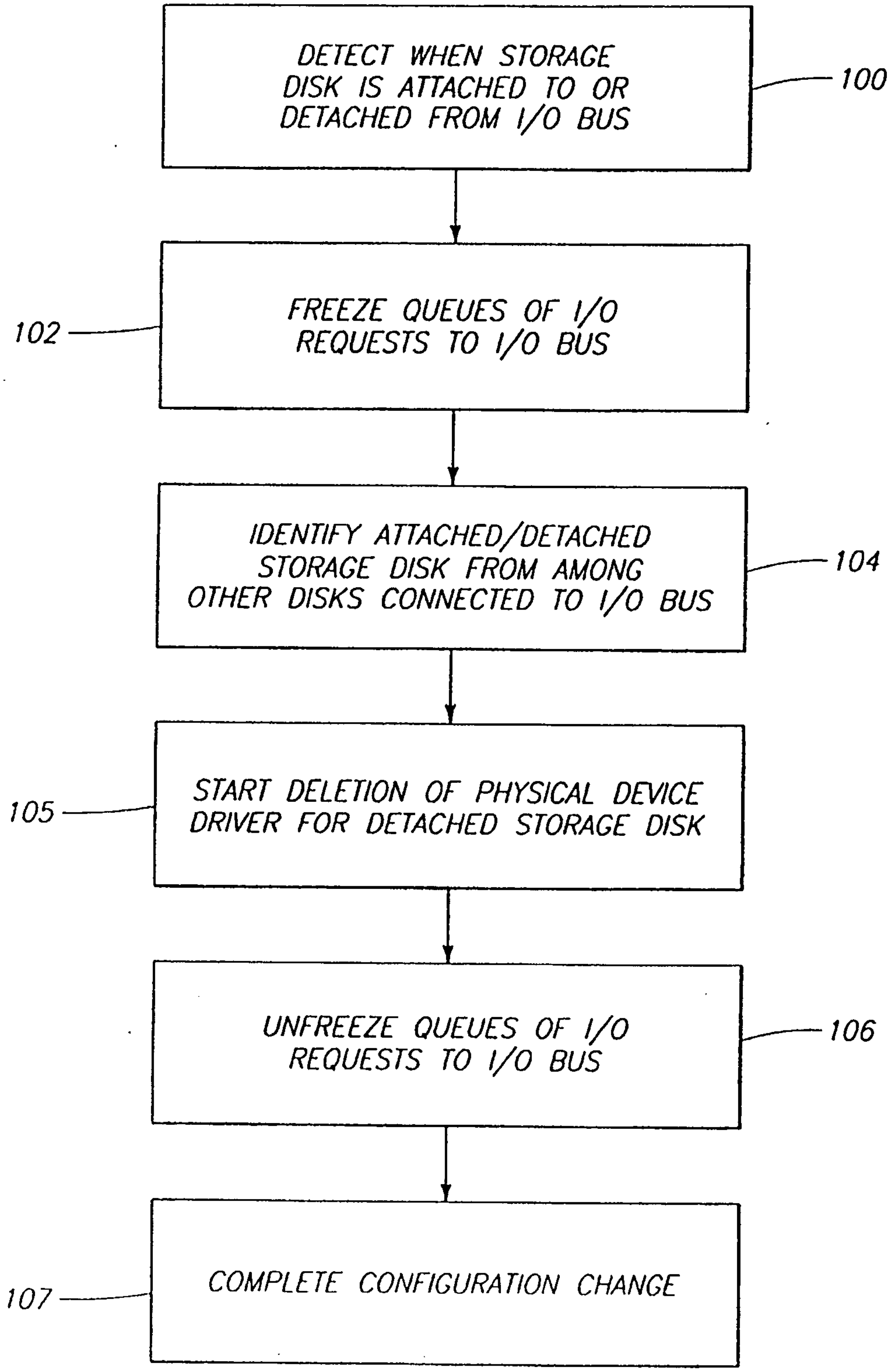
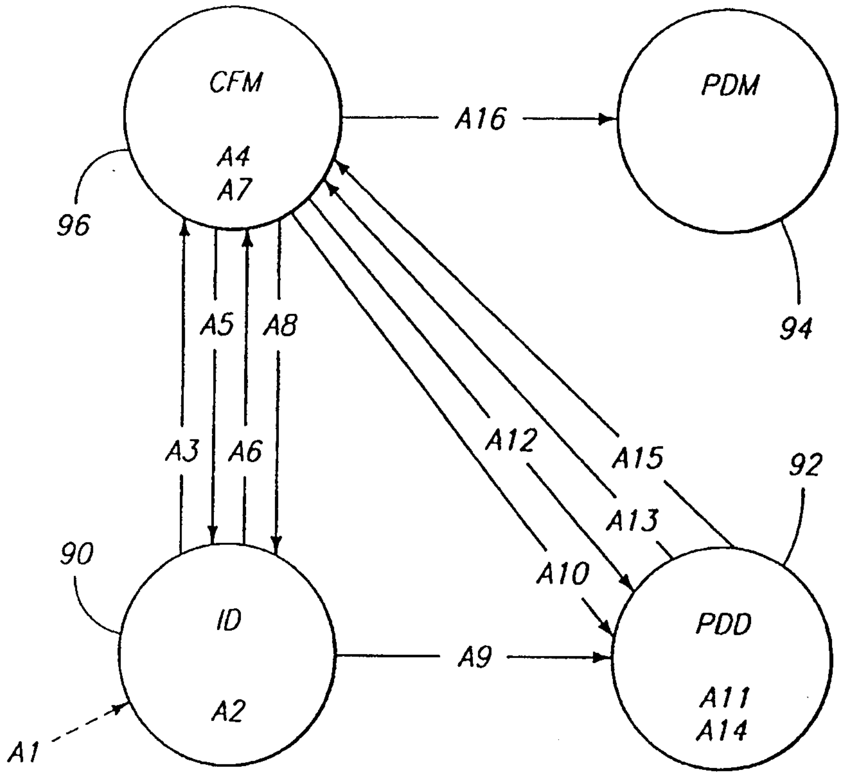

DETECT WHEN STORAGE DISK IS ATTACHED TO OR DETACHED FROM I/O BUS
100
FREEZE QUEUES OF I/O REQUESTS TO I/O BUS
102
IDENTIFY ATTACHED/DETACHED STORAGE DISK FROM AMONG OTHER DISKS CONNECTED TO I/O BUS
104
START DELETION OF PHYSICAL DEVICE DRIVER FOR DETACHED STORAGE DISK
105
UNFREEZE QUEUES OF I/O REQUESTS TO I/O BUS
106
COMPLETE CONFIGURATION CHANGE
107

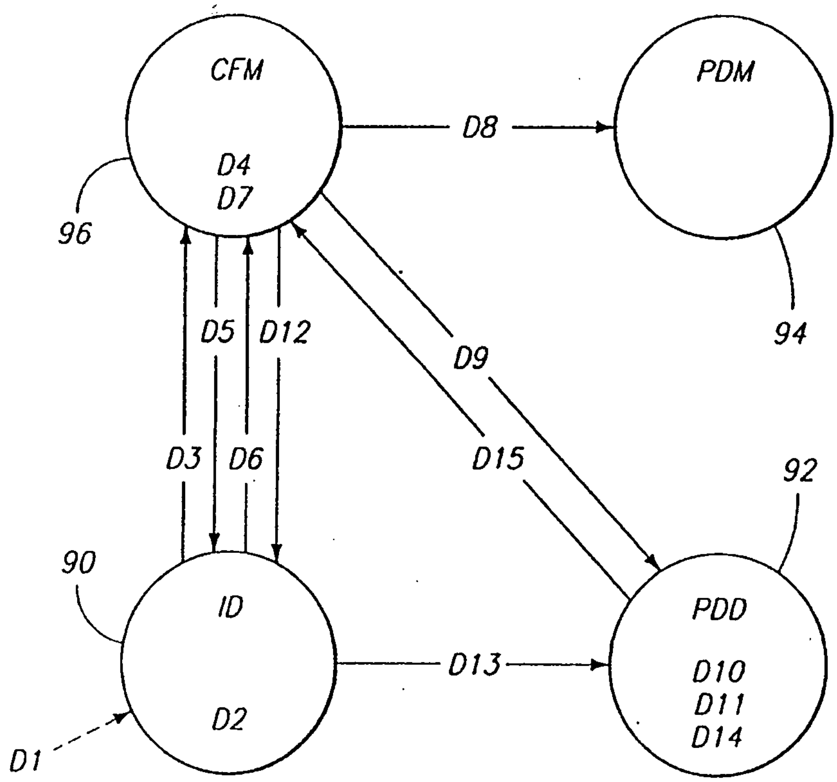
Fig. 4



TABLE

A1	BACKEND CHANNEL RESET DETECTED
A2	FREEZE QUEUES, ACCEPT I/Os BUT DO NOT PROCESS
A3	NOTIFY CFM OF BACKEND CHANNEL RESET
A4	SCAN FOR STORAGE DISKS
A5	REQUEST NEXT DISK INQUIRY DATA
A6	RETURN NEXT DISK INQUIRY DATA
A7	DETECT NEWLY ATTACHED STORAGE DISKS
A8	NOTIFY ID TO UNFREEZE QUEUES
A9	FLUSH QUEUE WITH HOT RESET STATUS
A10	CREATE PDD FOR NEWLY ATTACHED STORAGE DISKS
A11	INITIALIZE PDD
A12	REQUEST SPIN UP OF NEWLY ATTACHED STORAGE DISKS
A13	REPORT VENDOR UNIQUE SERIAL NUMBER
A14	SPIN UP AND CONFIGURE NEWLY ATTACHED STORAGE DISKS
A15	CALLBACK WHEN SPIN UP AND CONFIGURATION ARE COMPLETE
A16	ASSOCIATE NEW PDD WITH APPROPRIATE PDM

Fig 5



TABLE

D1	BACKEND CHANNEL RESET DETECTED
D2	FREEZE QUEUES, ACCEPT I/Os BUT DO NOT PROCESS
D3	NOTIFY CFM OF BACKEND CHANNEL RESET
D4	SCAN FOR STORAGE DISKS
D5	REQUEST NEXT DISK INQUIRY DATA
D6	RETURN NEXT DISK INQUIRY DATA
D7	DETECT REMOVED STORAGE DISKS
D8	PDM MISSING, STOP SUBMITTING I/Os TO PDD
D9	DESTROY PDD FOR REMOVED STORAGE DISKS
D10	CHANGE STATE TO DELETE-IN-PROGRESS
D11	FLUSH WAITING QUEUE
D12	NOTIFY ID TO UNFREEZE QUEUES
D13	FLUSH QUEUE WITH HOT RESET STATUS
D14	WHEN STATE IS DELETE-IN-PROGRESS AND ACTIVE QUEUE IS EMPTY, DO D15
D15	DELETE COMPLETE

FIG 6

European Patent Office

EUROPEAN SEARCH REPORT

Application Number
EP 95 11 4570

DOCUMENTS CONSIDERED TO BE RELEVANT

Category	Citation of document with indication, where appropriate, of relevant passages	Relevant to claim	CLASSIFICATION OF THE APPLICATION (Int.Cl.6)
A	WO-A-89 10594 (AMDAHL) * page 30, line 23 - page 31, line 32; figure 5 * ---	1,5,7,9	G06F15/16 G06F13/10
A	IBM TECHNICAL DISCLOSURE BULLETIN, vol. 37, no. 4b, April 1994, NEW YORK US, pages 391-400, XP002003402 "Concurrent Maintenance Direct Access Storage Device for Computer Systems" * page 391, paragraph 1 - page 391, paragraph 5 * * page 393, paragraph 3 - page 395, paragraph 3; figures * ---	1,5,7,9	
A	EP-A-0 505 792 (IBM) * page 4, line 10 - page 5, line 9; figures 1,2 * * page 18, line 54 - page 19, line 13 * -----	1,5,7,9	

TECHNICAL FIELDS SEARCHED (Int.Cl.6)

G06F

The present search report has been drawn up for all claims

Place of search	Date of completion of the search	Examiner
THE HAGUE	20 May 1996	Gill, S

CATEGORY OF CITED DOCUMENTS

X : particularly relevant if taken alone
Y : particularly relevant if combined with another document of the same category
A : technological background
O : non-written disclosure
P : intermediate document

T : theory or principle underlying the invention
E : earlier patent document, but published on, or after the filing date
D : document cited in the application
L : document cited for other reasons

& : member of the same patent family, corresponding document

PHD Computer Consultants Ltd
... *Windows Driver Model Article*

Last modified: 16 June 1999.

Approximately 4500 words.
This article was originally written for EXE Magazine and appeared in its January 1999 issue.

The source code for the HilmWdm example driver discussed in this article is available in the PHD download area. Download hiimwdm.zip and unzip in directory C:\EXEWdm.

You may be interested in reading Chris Cant's earlier article on writing Windows NT 3.51 and NT4.0, Writing Windows NT Device Drivers. Many of the techniques described in this article are still used in WDM device drivers.

Chris Cant's book on WDM device drivers is available now.

Check out PHD's other device driver resources. PHD sells its DebugPrint software that lets you include formatted print trace statements in your driver code, and a simple general purpose driver PHDIo.

Windows Driver Model (WDM) Device Drivers

Introduction

The **Windows® Driver Model (WDM)** lets you write a common device driver for Windows 98 and Windows 2000 (NT 5) - for some types of device.

This article will look at the development of a small virtual WDM device driver to illustrate the main principles without getting bogged down in the details. I mainly concentrate on giving an overview of the technology, rather than the minutiae of each line of code. Welcome to acronym-land.

WDM is an enhanced form of the NT 3.51 and NT 4 kernel-mode device driver

Glossary	
API	Application Programmer Interface
COM	Common Object Model
DDK	Driver Development Kit
FDO	Functional Device Object
GUID	Globally Unique Identifier

model. The main structural changes are the addition of **Plug and Play** (PnP), **Power management**, **Windows Management Interface** (WMI) and **Device Interface** support. These features are described in this article, along with details of the necessary development environment.

However, almost of more significance, is the provision of a series of **class drivers**. A class driver does the bulk of the work for a specific area of functionality. There are class drivers for the Universal Serial Bus (USB), the IEEE 1394 (Firewire) bus, streaming devices and Human Interface Devices (HID). More on these later.

You still need to write a separate driver which calls the appropriate class driver. For example, you can write a driver to talk to a new USB device through the USB class driver using relatively straightforward USB Request Blocks (URBs), ie without having to worry about the details of talking to a bit of hardware. For HID devices, you do not even need to write a kernel-mode driver at all, as you can use Win32 HID client functions to access the device.

There are two ways of customising class drivers. The first is to write a **filter driver**, which can slot in above or below a class driver. Alternatively, **minidrivers** in various shapes and form can be used. The class driver does all the general processing while the minidriver just communicates with a specific type of hardware. For example, the Windows HID class driver is paired with the hidusb minidriver to talk to HID devices on the USB bus. You could, for example, write a new HID minidriver to provide an interface to the serial port version of your product.

Each class driver has an appropriate specification. The USB class driver responds to several internal IOCTLs, one of which is to process a URB. As another example, a HID minidriver must register certain call-backs to work with the HID class driver. So you need to consult the relevant class documentation in the Driver Development Kit (DDK).

HID	Human Input Device
IEEE 1394	100Mbps+ serial bus, neé Firewire
INF	Installation information file
IOCTL	I/O Control Code
IRP	I/O Request Packet
ISA	Industry Standard Architecture PC bus
mof	WMI class file
MSDN	Microsoft Developer Network
NT	New Technology
PDO	Physical Device Object
PnP	Plug and Play
SDK	Software Development Kit
URB	USB Request Block
USB	12Mbps Universal Serial Bus
VC++	Visual C++
W2000	Windows 2000 (neé NT 5)
W98	Windows 98
WBEM	Web-based Enterprise Management
WDM	Windows Driver Model
WMI	Windows Management Instrumentation

Microsoft has managed to sneak COM Globally Unique Identifiers (GUIDs) into driver development. You can use these to define a private interface that identifies your particular device. GUIDs are sometimes also used by minidrivers to identify what facilities are supported.

Unfortunately, you will not be able to port your old NT device drivers to Windows 98 with no work. WDM relies on Plug and Play for resource assignment, which was not available in these old drivers. Some people will still need to write VxDs for Windows 98, and others will need to maintain their NT specific kernel-mode drivers for Windows 2000, eg for video drivers. See my earlier article in EXE magazine, October 1997 for details of NT kernel-mode driver development. In fact, I recommend that you re-read this article as it covers many concepts used here.

Currently it is not exactly clear whether WDM device drivers are binary compatible between W98 and W2000. The W98 DDK says that drivers can be compatible, while the W2000 DDK says that they are not necessarily binary compatible. The DDKs do claim that they should be source level compatible. However, for USB drivers, there are some facilities available in W2000 that are not available in W98.

Driver Development

You will need a Microsoft Driver Development Kit (DDK) or two to build drivers. However I have bundled a copy of the built driver with the source code, so you can try out the example driver without having a DDK. The DDKs are available for lengthy download online, but most driver developers want to have an MSDN Professional subscription to get the DDKs and the useful Platform SDK on CD, as well as the latest beta test versions of Windows, etc.

You can buy two different types of development tool to help you write drivers. The first type lets you control a general purpose driver in user-mode to handle many standard types of I/O. The second type gives you a C++ framework to base your driver on, with many useful classes and examples. Compuware Numega, BlueWater Systems, Inc. and Jungo provide products of both types. PHD sells general purpose drivers. Finally, Open Systems Research sells OSR DDK, a debugging aid that logs all your DDK function calls so that they can be viewed with their OSRTracer program.

The example driver comes with a suitable VC++ Visual Studio 97 workspace. The Makefile project assumes that you have the Windows 98 DDK in its default location of `C:\98DDK` and that the example source base directory is `C:\EXEWdm`. Alter the project settings if you use different directories.

The W2000 DDK provides the **WinDbg** tool to let you do source level debugging between two W2000 computers.

Resources	
Windows 98 DDK • Dr Iver	www.microsoft.com/ddk
MSDN • Windows 98 DDK • Windows 2000 DDK • Platform SDK	msdn.microsoft.com
Compuware Numega • SoftICE debugger • DriverAgent • DriverWorks	www.compuware.com/driverzone www.vireo.com www.numega.com
BlueWater Systems • WinDK • WinRT	www.bluewatersystems.com

However, the equivalent **SoftICE** tool from Compuware NuMega can be used on just one PC. For simple formatted print trace statements you can use PHD's **DebugPrint** software.

Jungo • WinDriver • KernelDriver	www.jungo.com
Open Systems Research • OSR DDK	www.osr.com
PHD • DebugPrint trace tool • PHDIo General purpose driver	www.phdcc.com/debugprint www.phdcc.com/phdio

HilmWdm example

Our device driver is called **HilmWdm** and the download zip hilmwdm.zip contains the source and built driver. Unzip the file in a directory called `C:\EXEWdm`.

The **HilmWdm** driver will run in Windows 98 and Windows 2000 without any new hardware. To test its operation, it implements a 4 byte shared memory buffer that can be read and written from a Win32 program. The "checked" build version also makes the buffer bytes available for inspection through WMI in W2000.

The `sys` directory contains the **HilmWdm** driver code. Table 1 lists the source code files and the files needed to build and install the driver. The "free" release version of the driver ends up in `OBJ\i386\free\HiImWdm.sys` with the "checked" debug version in `OBJ\i386\checked\HiImWdm.sys`

The `exe` directory contains a small test console application. Note that the VC++ 5 Setup API header and library is seriously out of date and will cause a compile to fail. The project is set up to use the `C:\98DDK` versions of these files. The W2000 DDK and Platform SDK versions are even newer.

Table 1	
HilmWdm source files	
file	**description**
HilmWdm.h	Header
Init.cpp	Driver Initialisation
Pnp.cpp	Plug and Play Power Management
Dispatch.cpp	Read, write, etc.
WMI.cpp	WMI handling
..\guid.h	GUID definitions
HilmWdm.rc	Version resource
HilmWdm.mof	WMI class definition
Build and installation files	
HilmWdmfree.inf	Free build INF

The W98 DDK does not currently include the WMI headers and libraries. So I have made the free build WMI-free. You can compile the checked build with WMI under W2000. The W98 DDK does not have the **mofcomp** tool so I have removed this compile step.

I would none-the-less have expected the checked WMI build to run under W98, as it is supposed to support WMI. (To install WBEM WMI, in `Add/Remove Programs, Windows Setup tab, Internet Tools`, check the Web-based Enterprise Mgmt box.) The Microsoft WBEM implementation seems to have been updated to include a WMI namespace that was not there before. Perhaps this update only runs properly in Windows 2000.

HilmWdmchecked.inf	Checked build INF
SOURCES	List of files to build
..\BuildDrvr.bat	Build batch file
makefile.inc	mof compile and Post build steps

Install and Test

To install the driver, go to the Control Panel and select "Add New Hardware" or "Hardware wizard". Opt to select the hardware from a list. Select "Other devices" and "Have Disk". Browse to `c:\EXEWdm\sys` and install the "HilmWdm Example, free build, without WMI" driver. The driver is now copied to the Windows `system32\drivers` directory and the installation INF file is copied to one of the Windows `INF` directories.

The **HilmWdm** device should now appear in the Device Manager "Other devices" category. If you rummage around the registry you will find references to the driver, the device and the device interface that have been installed. The Windows Unknown GUID {4D36E97E...} appears in the first two cases, while the device interface uses the HilmWdm GUID {87472BA0...}. Be warned that W98 and W2000 use slightly different registry structures.

To test the driver, run `TestWDM.exe` in the `C:\EXEWdm\exe\Release` directory. **TestWDM** opens a handle to the first HilmWdm device and reads the buffer. The first time you run it, the buffer will probably have a value of zero. However, the next time it should be storing 0xABCDEF01, left over from the last write.

TestWdm goes on to write 0xABCDEF01 to the buffer and checks that this value can be read. It then checks that an incorrect write fails and finally closes the device handle.

Initialisation

The driver's main entry point is the `DriverEntry` routine in `init.cpp`. The main job here is to set up the series of call-back routine pointers shown in Table 2 so that the driver can be called again when appropriate. Some calls originate in the kernel. A Win32 program accesses the device as if it were a file, and these requests end up as driver calls as well.

A driver works primarily by processing I/O Request Packets (IRPs), so call-backs are defined for each of the IRP major functions that HilmWdm supports. Some of the IRPs come in different forms, eg the Plug and Play IRP_MJ_PNP has a IRP_MN_START_DEVICE minor code which indicates that this IRP requests you to start the device.

This example does not cope with IRP cancelling.

Table 2

HilmWdm call-backs

call-back	description
AddDevice	PnP Add new device
Unload	Driver unload
IRP_MJ_CREATE	Win32 CreateFile
IRP_MJ_CLOSE	Win32 close file
IRP_MJ_POWER	Power management
IRP_MJ_PNP	Plug and Play
IRP_MJ_READ	Win32 reads
IRP_MJ_WRITE	Win32 writes
IRP_MJ_DEVICE_CONTROL	Win32 IOCTLs
IRP_MJ_SYSTEM_CONTROL	WMI

Plug and Play

Plug and Play (PnP) makes it easier for users to insert new devices into a computer as there should be no complicated hardware addresses to set up. Instead a Plug and Play device should be configurable in software. For a low level bus driver, its PnP resources are the different sets of IRQs, I/O space registers and memory-mapped addresses that it supports. Most bus types define their own scheme for making devices configurable. Some ISA devices are PnP configurable as well.

Device Stack

PnP builds a device stack from the bottom up. I shall now describe how the right drivers are found for USB devices. Details of the USB terminology are given later.

PnP uses enumerator drivers and arbiters to find devices and allocate resources to them. The root enumerator might find a PCI bus. The PCI bus enumerator might find a USB host controller. The USB host controller class/miniclass drivers are loaded, and, above them, the USB Hub driver for the embedded USB root hub. The USB hub then enumerates the USB bus.

When a USB device is first plugged in, it appears at the USB default address of zero. The act of inserting the device notifies the USB hub of a new device. It retrieves the USB device's descriptors and in due course allocates it a proper USB address.

Each device has a ***Hardware Id*** which is used to identify the appropriate driver. In the USB context, the ***Hardware Id*** is formed using the vendor's id and the product id. A fallback ***Compatible Id*** is also made available in case an appropriate driver is not found. For USB, this is the class and sub-class of the device. For example, Windows recognises the "HID keyboard USB device" class so that it can be used straight away without a vendor-supplied driver.

A driver's INF installation file lists a driver for each ***Hardware Id*** and ***Compatible Id*** supported. The INF file lists the files that need to be copied and the registry entries that should be made. W2000 specific sections are used to set up the driver service registry entry and any error logging features.

When a device is added, it is put at the top of a stack of devices, for example, above the HID class/minidriver driver pair, the USB class driver and the PCI bus driver. Any I/O requests are sent to the top of the device stack. Each layer of the stack can do some processing on the request, or pass it down the stack. A driver can attach a completion routine to an IRP so that it can handle the IRP after it has been processed by the rest of the stack below it.

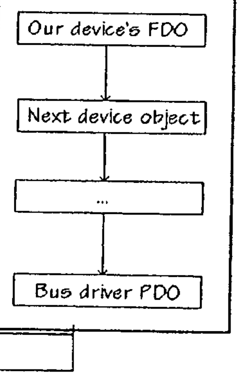
Note that a driver can have an "upper edge" interface which is totally different to the facilities it uses in the next lower driver. A device might have a streaming upper edge, but make calls to the USB system to do its job.

Device Objects

A WDM driver has to cope with three different types of device object as shown in figure 1. The bus driver object at the bottom of the device stack is called the Physical Device Object (PDO). Each WDM driver must make its own Functional Device Object (FDO) for each device. Finally, you must remember the next device down the stack. This is so you know where to pass a request when you need to send it for processing down the stack of drivers.

Figure 1

Stack of device objects



Listing 1 shows how HilmWdm's FDO is made in the `AddDevice` routine using the **IoCreateDevice** call. `AddDevice` then attaches to the top of the stack above the PDO using **IoAttachDeviceToDeviceStack**. The returned FDO device object has a pointer to some memory for us to use, the device extension. The HilmWdm device extension structure is defined in `HiImWdm.h`. Our `AddDevice` stores the PDO, FDO and the NextDevice in the device extension.

When our device is removed, our FDO is detached from the stack and deleted.

Listing 1

`AddDevice` routine Device object handling

```
// Create our Functional Device Object in fdo
status = IoCreateDevice(DriverObject,
        sizeof(WDM_DEVICE_EXTENSION),
        NULL,    // No Name
        FILE_DEVICE_UNKNOWN,
        0,
        FALSE,   // Not exclusive
        &fdo);

// Remember pdo and fdo in our device extension
PWDM_DEVICE_EXTENSION dx = (PWDM_DEVICE_EXTENSION)fdo->DeviceExtension;
dx->pdo = pdo;
dx->fdo = fdo;

// Attach to the driver stack below us
dx->NextDevice = IoAttachDeviceToDeviceStack(fdo,pdo);

// Set fdo flags appropriately
fdo->Flags &= ~DO_DEVICE_INITIALIZING;
fdo->Flags |= DO_BUFFERED_IO;
```

Plug and Play States

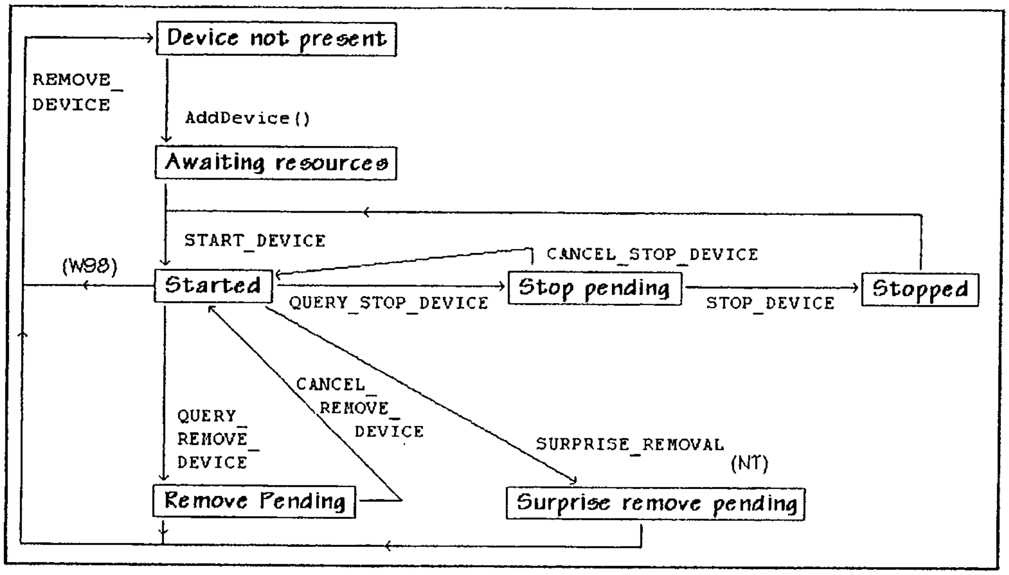
A PnP driver goes through several different states as devices are added, removed or stopped to allow for resource reallocation.

When a new device is loaded, the PnP Manager eventually finds the relevant driver for a device. The driver is loaded and its `DriverEntry` routine called.

Figure 2 shows the different states that a PnP device can then go through. First a driver's `AddDevice` routine is called so that the driver can create its FDO and attach it to the stack. However, do not talk to your device yet!

Figure 2

Plug and Play States



When your driver receives a IRP_MJ_PNP request with the IRP_MN_START_DEVICE minor function code, all the relevant hardware resources have been assigned. Normally you must pass this request down the driver stack first so that the bus driver can process it, ie make the device available to you. Your IRP completion routine can talk to your device as the IRP travels back up the driver stack. Indeed, you may well want to send more requests down the stack to configure your device, before finally completing the original start device IRP.

Similar considerations apply to stopping or removing devices, ie do any of your work to stop the device before you send the IRP down the stack.

The other PnP states are used to cope with device removal requests and stop requests. Both these have query IRPs to let you reject the request for the moment. You might say "no" if you have a read or write request in progress. You should store any further ordinary I/O requests in a queue while a device is stopped, and reject them if a device is removed. If a user brutally unplugs a device then you must cope with a IRP_MN_REMOVE_DEVICE in Windows 98 or a IRP_MN_SURPRISE_REMOVAL in Windows 2000.

Plug and Play Flags

Proper PnP handling requires the use of various flags, etc., in the FDO device extension. You will usually need a flag to indicate that the device has been started. Check this flag before performing any I/O requests.

I/O operations can be in progress when a device is removed or stopped. You must ensure that the remove or stop request waits until the current I/O is cancelled or finished. Drivers usually do this by having a usage count and StoppingEvent event in each device extension. The usage count is incremented atomically when an I/O operation is started and decremented when it completes. A remove request checks that the usage count is zero. Otherwise it waits for the StoppingEvent to be signalled when an I/O operation completes.

While your device is stopped for resource reallocation, you should hold any incoming I/O requests in a queue for processing when the device is restarted.

HilmWdm currently only provides minimal PnP support. It responds to `AddDevice` and REMOVE_DEVICE calls, ie to make and delete its FDO. It does not use any PnP flags to queue IRPs during stops nor check remove requests.

Device Interfaces

A device must be accessible to the kernel or Win32 code for it to be of any use. The old NT device driver model uses explicit symbolic links to provide a name which a Win32 application could open.

While this technique is still available, WDM lets you use device interfaces instead. A device interface uses a GUID to identify the interface that a driver implements. In HilmWdm, we just use the `WDM_GUID` GUID to identify the fact that we are a HilmWdm device. In other cases, a driver might use a standard GUID to indicate that it implements a particular COM interface.

Our `AddDevice` routine calls **IoRegisterDeviceInterface** to register the link between `WDM_GUID` and our FDO. It then has to enable it using **IoSetDeviceInterfaceState**. When our device is removed, the `WdmPnp` routine disables the device interface.

IoRegisterDeviceInterface actually makes a symbolic link to our device. The actual link name is a long string which includes our GUID. Win32 programs like our TestWdm use various **SetupDi...** functions (such as **SetupDiGetClassDevs**) to find all devices which support a particular GUID. Eventually it can get the symbolic link name which it can pass to **CreateFile** to open a handle to our HilmWdm device.

Power Management

Device drivers play an important part of Power Management. The idea is to have shorter start-up and shutdown times by not turning off the computer completely. Power Management policies can also help to conserve battery life and might result in quieter running of the computer.

There are six system power states defined, S0 to S5, with S0 being fully on and S5 shutdown. Each device can be in one of four power states called D0 to D3, with D0 fully on.

A device might decide to reduce its own power level, eg if a disk has not been accessed for 5 minutes. Alternatively, the Power Manager can request that the whole system power down (it assumes that the system can always power up). A device that is sleeping (S1-S3) or hibernating (S4) can wake the computer up, eg if a modem receives an incoming call.

Power Management is done with the IRP_MJ_POWER IRP. The Power Manager uses the IRP_MN_QUERY_POWER minor code to see if a driver can go into a specified state, and then IRP_MN_SET_POWER to actually request that state. If you are going to fail a power request then complete the IRP immediately. Otherwise it is up to the bus driver to complete the power IRP.

Drivers like HilmWdm that do not implement a power policy should just pass the power IRPs down the stack. This means calling **PoStartNextPowerIrp**, **IoSkipCurrentIrpStackLocation** and then **PoCallDriver**. Note the call to **PoCallDriver** rather than **IoCallDriver** to call the next driver.

If a driver does handle power IRPs then proceed as follows. The Power Manager initiates a IRP_MN_QUERY_POWER IRP to set a new system power state. If your device needs to change to a different device power state that is appropriate for the given system power state, then you need to send yourself a power IRP to do this. Yes, that's right, you call **PoRequestPowerIrp** to tell yourself to change power state. Once this IRP has completed its rites of passage, you can pass on (or complete) the original system power IRP.

If you get an I/O request while powered down, you must send a set power IRP and wait for its completion, before handling the request.

WMI

Windows Management Instrumentation (WMI) lets administrators tend your device. So your driver must provide information and events to user-mode applications. Methods in the driver can be invoked.

WMI is a part of the Web-Based Enterprise Management (WBEM) initiative. The Win32 implementation of WBEM has various providers of information, giving access to the registry, the NT event log, Win32 information and WMI.

For WMI, you can define your own custom data and event blocks in a C++ class-like file, compiled into a resource using the **mofcomp** tool. Each WMI block is identified by a GUID. A WBEM Object Browser tool lets you view all the WMI blocks on a computer, or across the network. Alternatively, custom user mode programs can be written to access WMI, either using Java APIs or COM ActiveX interfaces.

Windows 2000 drivers can and should still send NT events to the system log. However WMI is supposed to be available in Windows 98, giving WMI broader appeal. As stated above, at the moment I have only got the WMI to run in Windows 2000.

The HilmWdm code in `wmi.cpp` shows how to provide a custom WMI data event called `HiImWdmInformation`. This contains the first ULONG from the read/write common buffer and the symbolic name for the device interface.

HilmWdm calls **IoWMIRegistrationControl** when a device is added or removed. It sets up a WMILIB_CONTEXT structure in its device extension with various call-backs. When the IRP_MJ_SYSTEM_CONTROL IRP is received, HilmWdm calls **WmiSystemControl** to do preliminary processing. The `QueryWmiRegInfo` call-back is called to register our WMI data block. `QueryWmiDataBlock` is called to return the actual WMI data.

USB

Finally, let's have a brief overview of USB and HID device drivers.

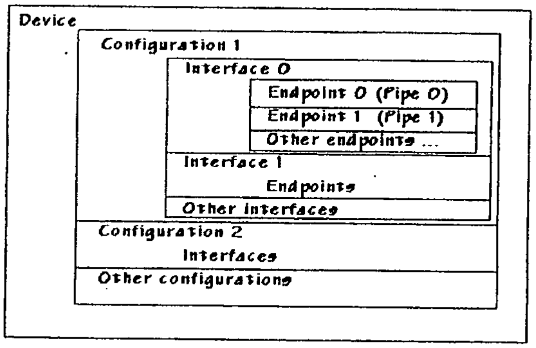
USB is for lowish speed devices. USB is a half-duplex 12 Mbps serial bus, with 5V lines to provide a small amount of power to basic devices. The USB data bits are grouped into 1ms frames, which are the basis of bandwidth allocation. Hub devices allow further function devices to be plugged in, even when switched on. A new PC usually has one root hub with 2 USB downstream ports.

Figure 3 shows the logical structure of a USB device, with endpoints grouped into interfaces and configurations. Almost all USB devices have just one configuration and most have just one interface. Windows needs a separate USB client driver for each interface.

Each endpoint can transfer one of four types of data: control, interrupt, bulk and isochronous. A connection to an endpoint is called a pipe. Transfers on endpoint 0 (the Default Pipe) have a standard format.

Figure 3

USB Logical Structure



Each USB device has a series of descriptors which describe its logical structure. The presence of additional class descriptors indicates that the device is of a certain standard type, eg printer, HID, hub, display etc. If Windows detects a USB device with a HID descriptor, then it automatically fires up the HID system drivers to interrogate the device. I guess that Windows will support other device classes in due course.

In the mean time, to control your USB device, you will need to write a WDM USB client kernel-mode driver. The Windows USB class driver is controlled using a series of internal IOCTLs. The most useful of these allows you to send off USB Request Blocks (URBs) for processing. There are currently 39 different URB operations that you can request, allowing you to get descriptors, perform all the different transfer types, etc.

HID

A Human Input Device (HID) is not some gruesome force-feeding contraption. Instead it is an abstract model for most types of input device that people could use to control their computers. HID lets you output information as well, eg to set the LEDs on a keyboard. Apparently there is a standard HID set of controls for a magic carpet!

Windows has built in support for various HID devices, eg keyboards and mice. Microsoft seems to be using HID more and more to get user input, as it provides an abstraction layer above the actual hardware interface. If both a HID keyboard and an old PC keyboard are attached then you can use them both.

Although HID was born as a USB extension class, it can now stand alone as long as the right HID descriptors are presented to the HID class driver. A USB minidriver is supplied as standard, but you can write other minidrivers if you wish.

Figure 4

Possible HID driver stack

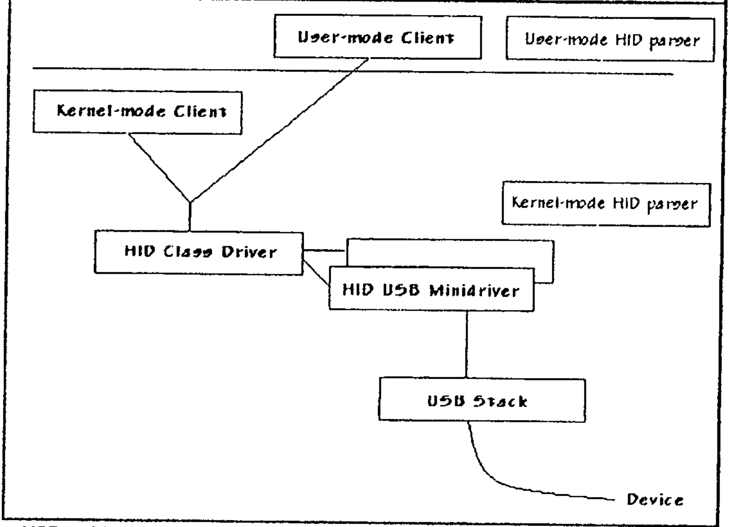


Figure 4 shows one possible HID driver stack configuration. A USB HID device is plugged in. The USB HID minidriver provides the interface between the USB and HID class drivers. HID clients can either run in the kernel (like the Windows keyboard drivers, `Vkd.vxd/kbdhid.vxd` or `Kbdclass.sys/Kbdhid.sys`) or in Win32 user mode.

A HID device describes its capabilities primarily in a Report Descriptor. Input, Output and Feature reports are described. Each report consists of a series of bit or data controls, possibly grouped into collections. Each control or collection has a "usage", a standard definition of what it does. A keyboard must produce exactly the right reports for it to be recognised by Windows and your BIOS.

In the general case, a HID client uses the Windows HID parsing routines to determine what usages a device is capable of producing. When it received an actual HID report it uses more Windows routines to determine what values were returned.

Table 3 shows a brief summary of the standard HID keyboard reports. The modifier keys are used for left Ctrl, left Alt, etc. The scan codes represent the keys that are pressed simultaneously. A keyboard report is generated whenever a key is pressed or released.

Table 3
HID keyboard report definition
Keyboard Collection Input: 8 single-bit modifier keys Output: 5 single-bit LEDs Output: 3 single bits for padding Input: 6 data bytes for scan codes

Conclusion

You must write a WDM driver if you want to support some of the latest technologies like USB, HID and IEEE 1394. However you still need to write separate video drivers for W98 and W2000.

Supporting Plug and Play and Power management does make a driver more complicated. However this extra work is usually more than offset by having standard class drivers to do most of the detailed hardware interactions for you. And it is certainly nice to be able to write one driver that works in Windows 98 and Windows 2000.

Author

Chris Cant
Director, PHD Computer Consultants Ltd
Email: sales@phdcc.com
Web: www.phdcc.com